The Paralax Infrastructure: Automatic Parallelization With a Helping Hand

The Paralax Infrastructure: Automatic Parallelization With a Helping Hand

Hans Vandierendonck
Dept. of Electronics and Information Systems
Ghent University
Belgium
hvdieren@elis.ugent.be

Sean Rul
Dept. of Electronics and Information Systems
Ghent University
Belgium
srul@elis.ugent.be

Koen De Bosschere
Dept. of Electronics and Information Systems
Ghent University
Belgium
kdb@elis.ugent.be

ABSTRACT

Speeding up sequential programs on multicores is a challenging problem that is in urgent need of a solution. Automatic parallelization of irregular pointer-intensive codes, exemplified by the SPECint codes, is a very hard problem. This paper shows that, with a helping hand, such auto-parallelization is possible and fruitful.

This paper makes the following contributions: (i) A compiler-framework for extracting pipeline-like parallelism from outer program loops is presented. (ii) Using a light-weight programming model based on annotations, the programmer helps the compiler to find thread-level parallelism. Each of the annotations specifies only a small piece of semantic information that compiler analysis misses, e.g. stating that a variable is dead at a certain program point. The annotations are designed such that correctness is easily verified. Furthermore, we present a tool for suggesting annotations to the programmer. (iii) The methodology is applied to auto-parallelize several SPECint benchmarks. For the benchmark with most parallelism (hmmer), we obtain a scalable 7-fold speedup on an AMD quad-core dual processor.

The annotations constitute a parallel programming model that relies extensively on a sequential program representation. Hereby, the complexity of debugging is not increased and it does not obscure the source code. These properties could prove valuable to increase the efficiency of parallel programming.

Categories and Subject Descriptors

Software [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; Software [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Design, Performance

Keywords

Semi-automatic parallelization, semantic annotations

To appear in the 2010 International Conference on Parallel Architectures and Compilation Techniques, September 11 - 15, 2010, Vienna, Austria.

1. INTRODUCTION

Parallel programming has been with us since the advent of computing. Until recently, parallelizing programs was not always worth the effort as single-threaded performance doubled every 18 to 24 months; a consequence of technology advances (scaling, frequency increase) and architectural improvements of processors. Since 2004, however, the economics have changed: by necessity, processor manufacturers have turned to multi-core processors where single-thread performance increases "only" by about 20% per year. Due to the proliferation of multi-core processors, all application domains are now confronted with thread-level parallelism, even those that are not easily amenable to such parallelism.

Many programming models exist to create parallel programs, ranging from low-level models (e.g. POSIX threads [6]), to higher-level models (e.g. OpenMP, MPI [31], Cilk [14]), to productivity languages (e.g. X10 [40], UPC [18]) and to domain-specific approaches (e.g. StreamIt [15]). Each of these languages matches particularly well to specific program structures, most often scientific computing or streaming operations. Irregular pointer-based applications are not targeted, yet parallelization of such codes is also mandatory.

The languages cited above are explicitly parallel: the programmer is burdened with the tasks of explicitly identifying parallelism, transforming the program and debugging the performance to verify the utility of the effort. In general, explicit parallel programming languages complicate program maintenance and debugging. Interactive parallelization tools [3, 20, 21, 23] aid the programmer with the parallelization and thread mapping tasks, although these tools still require significant effort from the programmer.

Ideally, programs are automatically parallelized. Research of the '80s and '90s has resulted in successful parallelization of DOALL and DOACROSS loops [5, 22, 30]. These techniques apply very well to array-based languages such as Fortran; but little success was obtained on irregular pointer-intensive C codes.

In this paper, we explore the semi-automatic parallelization of irregular pointer-intensive C codes. Hereto, we use an implicit parallel programming methodology [19], which assumes that the programmer is aware that the program will execute on parallel hardware, but he does not have to write explicitly parallel programs. Rather, an auto-parallelizing compiler turns the sequential program into a parallel one. This gives the benefit of exploiting performance improvements due to parallelism while writing code in a sequential programming model.

This paper makes the following contributions in order to make the implicit parallel programming approach work on pointer- and control-intensive C codes.

```
// Global variables
char * block;      int last;      short * szptr;

void compressStream(int fd_in, int fd_out) {
    // ... setup, allocate memory for block and szptr ...
    while(1) {
        loadAndRLESource(fd_in);      // initialize block, last
        if(last == -1)
            break;
        doReversibleTransformation(); // use block and last, may modify block
        generateMTFValues();          // initialize szptr, use block and last
        sendMTFValues(fd_out);        // use szptr and last
    }
}
```

Figure 1: Simplified code for bzip2 compression.

1. We present the Paralax compiler, an auto-parallelizing compiler for coarse-grain loops operating on whole data structures. This approach works well as alias analysis succeeds at grouping memory references per data structure and loop bodies are quite large. In contrast, parallelization of fine-grain loops in C programs has not been successful as this depends too much on accurate intra-data structure alias analysis and loop bodies are too small.

2. We present LWPM, a light-weight programming model based on annotations. The annotations detail semantic properties of functions, variables and function arguments. An important annotation is, e.g., the KILL annotation which states that a variable is dead at a certain program point. The unique property of this programming model is that it does not explicitly steer parallelization; parallelization follows automatically from the increased accuracy of compiler analysis.

3. We present methods for debugging the correctness of the annotations. As the annotations are not provable by our auto-parallelizing compiler (otherwise they would be redundant), we propose functionality to turn the annotations into code fragments that check their correctness during execution of the program.

4. We present a tool for proposing where to insert annotations in a program. Hereto, we capture dynamic dependence information during profiling executions and we compare it to statically determined dependences. The difference between the sets of dependences indicates where annotations may be applicable. Such a tool helps programmers to upgrade sequential programs to implicitly parallel ones.

The paper is structured as follows. Section 2 presents the compilation flow and the design decisions appropriate for parallelizing irregular pointer-intensive programs. Section 3 presents the light-weight programming model that conveys additional information to the compiler. Section 4 presents programmer tools to help the programmer with inserting annotations and to test the correctness of annotations. Next, we apply the techniques to benchmarks and provide numerical evaluation in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2. COMPILATION FLOW

We illustrate the compilation flow using a simplified version of the main compression loop in bzip2 as example (Figure 1). The code makes extensive use of global variables, in this case *block*, *last* and *szptr*. Many more variables appear in the real code but these are omitted for pedagogical reasons. The example code first allocates memory and initializes the global pointers. Then, it enters a main loop, consisting of four main stages as indicated by four function calls.

The loop can be parallelized as a parallel-stage pipeline: the *loadAndRLESource()* and *sendMTFValues()* functions carry dependences and must be executed sequentially, but the remaining function calls are highly parallel. In fact, these functions may be executing multiple times in parallel, each one operating on the data computed by a different loop iteration.

We illustrate below how the Paralax compiler combines several state-of-the-art algorithms to recognize parallelism in this code.

2.1 Memory Analysis

The first step of analyzing memory is to identify data structures. A data structure is identified by its type and a base pointer. Recognized types are primitive types such as integer types and floating point types. They can also be composite types such as a pointer to a type, a structure (an ordered collection of types) or an array (a repetition of a type). Types may also be undefined in cases where types cannot be accurately determined. We use a unifying shape analysis to determine the types of data structures, in particular Data Structure Analysis [28].

Example: Data Structure Analysis easily identifies the globals used in the program and can reconstruct from the code that *block* and *szptr* are used as pointers to heap-allocated arrays of type *char* and *short*, respectively.

2.2 Dependence Analysis

Dependence analysis tracks the pairs of statements or instructions that have dependences through data structures stored in memory. Hereto, algorithms based on use/def chains or static single assignment (SSA) may be used; the actual algorithm used is orthogonal to this paper. SSA however has some advantages, e.g. it simplifies privatization of data structures when generating multi-threaded code.

Applying SSA to memory variables has proved tricky due to the partial updates of data structures made by word-size stores to elements or fields. Roughly speaking, existing solutions range from applying SSA to individual words in memory [9, 27, 32] to full data structure phi-nodes as in Array SSA [13, 24]. In the first case, the granularity of the representation is too fine to facilitate full-data structure transformations such as privatization. In the latter case,

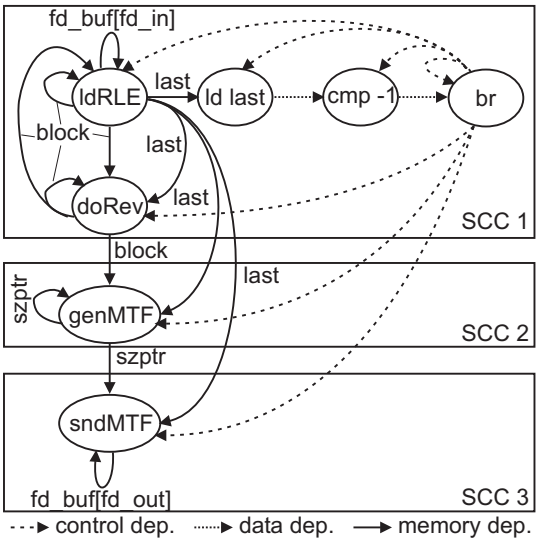

Figure 2: Program dependence graph for the bzip2 code.

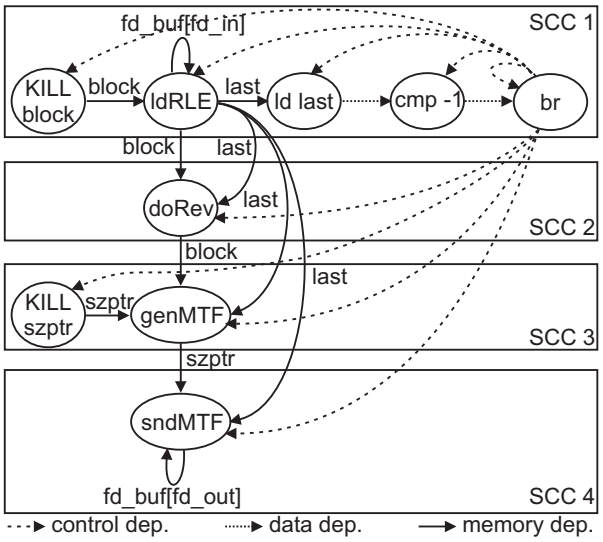

Figure 3: Program dependence graph after introducing KILL statements.

the merging effect of phi-nodes must be evaluated at runtime in order to model partial updates [24].

The Paralax compiler uses a mixture of full-data structure SSA and use/def chains. The idea is to create phi-nodes only when a data structure is fully defined and to use use/def-chains on all operations on the same SSA version.[1] This strategy avoids the complexity of Array SSA [24], i.e. runtime evaluation of merging phi-nodes, while providing the benefits of SSA that are most important in the present context.

Example: Dependence analysis is not quite precise. Figure 2 shows the program dependence graph (PDG) [12], a graph where each node represents an instruction and where edges represent control, data and memory dependences between instructions. The four nodes lined up vertically correspond to the four function calls, while the nodes on the top row correspond to the loop termination test and exit branch.

Dependence analysis conservatively assumes some non-existing dependences, in particular the dependence of *loadAndRLESource()* on *doReversibleTransformation()* and the fact that each function that initializes an array is also dependent on itself, i.e. there is a loop-carried dependence. In contrast, dependence analysis does know that the global *last* is re-initialized on every loop iteration as there are no loop-carried dependences on the *last* variable. The reason is that it is easy to see that a scalar is defined, but it is much harder to prove that every array element is defined.

2.3 Parallelization

We follow Allen and Kennedy in order to detect parallelism [2]. Parallelism is detected by computing the strongly connected components (i.e. cycles) on the PDG of a loop. Each strongly connected component (SCC) represents a group of instructions that are cyclically dependent. As such, they cannot be split across pipeline stages. The SCCs are clustered in pipeline stages using basic block execution frequencies and inter-SCC dependences in order to load balance the pipeline [33, 35]. Parallel-stage pipelines are possible when an SCC does not have a loop-carried dependence with itself. The compiler uses a static performance model to predict the speedup of parallelization. Only loops with significant speedups are parallelized.

The Paralax compiler also recognizes task parallelism outside of loops, a pattern that is similar to the OpenMP sections construct.

It can happen that a group of instructions is cyclically dependent on a particular data structure or variable, e.g. in a reduction or when manipulating I/O streams. Such groups of instructions are known as ordered sections [45]. The decoupled software pipelining model can handle such instructions when they occur in the loop body, but not when they are embedded in callee functions. We analyze whether ordered sections can be extracted from the callees and be executed out-of-line in the loop body. Hereto, the ordered sections are "queued up" together with the required input data. The ordered sections are then executed in original program order by reading elements from the queue and executing them in a sequential pipeline partition [45].

Example: The PDG of the example loop contains three SCCs (Figure 2). Each SCC has a loop-carried dependence, allowing to transform the code to a three-stage pipeline.[2]

2.4 What's Missing

We noted that there are a number of spurious dependences in the PDG of Figure 2, resulting from the fact that dependence analysis cannot determine precisely the last def of array elements. In particular, (i) it does not know the size of the arrays as they are heap-allocated and (ii) the arrays are not necessarily entirely overwritten on each loop iteration, although the programmer knows that array elements are used only if they have been defined in the same loop iteration. What we need is to convey this semantic information to the compiler to allow it to find more parallelism.

We propose several annotations to do just this (Section 3). One particular annotation is the KILL statement, the effect of which we describe here. The KILL statement tells the Paralax compiler that a particular data structure is dead at the point where the statement is placed. Dependence analysis picks up on this and assumes that the KILL statement defines fully the referenced data structure and that

[1] It is also feasible to use an SSA algorithm on individual words to represent accesses to the same version of a data structure.

[2] Note that the real bzip2 code contains many more dependences than the example. Consequently, the compiler cannot discover the pipeline in the real code.

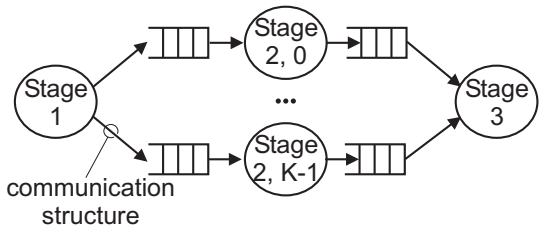

Figure 4: Mapping a parallel-stage pipeline to threads. Each circle corresponds to a thread executing code for one particular stage. Queues effect communication. The entity of communication is a communication structure that groups all variables passed between the pipeline stages.

it never uses any data structure. Hereby, use/def chain traversals terminate in KILL statements, decreasing the number of dependences.

Example: Two KILL statements are introduced at the beginning of the example loop body, one on *block* and one on *szptr*. The effect of this on the PDG is dramatic (Figure 3). We find (i) fewer loop-carried dependences, (ii) one additional SCC and (iii) the loop can be parallelized as a parallel-stage pipeline because SCC2 and SCC3 do not have loop-carried dependences. All we had to do to obtain this is to introduce a few KILL statements, information that is, in principle, clear to the programmer.

2.5 Code Generation

Now that we are happy with the degree of parallelism found and now that tasks have been identified and load balanced by the DSWP algorithm, we proceed with generating parallel code. For pipelines, each pipeline stage is mapped to a distinct thread which executes the stage for every iteration of the loop. For parallel-stage pipelines, the sequential stages are mapped to a distinct thread, while the remaining threads each execute iterations of the parallel stage. We consider only pipelines with at most one parallel stage. Figure 4 shows how stages are mapped to threads. Stage 2 is executed by K threads, resulting in a total of $K + 2$ threads. It is common that stages 1 and 3 perform relatively little work. This makes it worthwhile to generate code for more threads than the actual core count.

Communication and synchronization between pipeline stages is effected by means of a C-style struct called "communication structure". This structure gathers all variables that are passed between pipeline stages, including privatized data structures. The first stage of the pipeline is modified to set up a fresh structure for each iteration of the loop and to initialize live-in values. All references to variables passed between pipeline stages are rewritten to load and/or store to the structure in a similar way to DSWP code generation. When functions called from the loop access globals that are passed between pipeline stages, then these functions are also rewritten to access the copy of the globals in the communication structure. The final pipeline stage is rewritten to copy live-out values to the original program variables and to cleanup the communication structure.

The communication structure is passed between threads by means of queues. The structure is queued up only after a pipeline stage has fully executed. As the following pipeline stage can only start executing after retrieving the communication structure from the queue, the queues implement all the necessary synchronization.

We currently schedule loop iterations statically, i.e. we decide during code generation what thread executes what pipeline stage and what loop iteration of that stage. This strategy is optimal for pipelines, but may be sub-optimal for parallel-stage pipelines. Static scheduling was chosen because it allows us to generate faster code and to use only lock-free single-producer single-consumer queues. When communication intensity is low, the Paralax compiler uses the more expensive POSIX locks in order not to waste processing cycles on polling the queues. This allows us to generate more threads than the number of hardware cores when pipeline stages are imbalanced.

```
char *strtok(char * restrict REF MOD s,
  const char * restrict REF NOCAPTURE delim) RETALIAS(0);

char *strtok_r(char * restrict REF MOD NOCAPTURE s,
  const char * restrict REF NOCAPTURE delim,
  char ** restrict REF MOD NOCAPTURE KILL save_ptr)
  RETALIAS(0) STATELESS;
```

Figure 5: The function argument annotations applied to the strtok() and strtok_r() C library functions.

3. PROGRAMMING MODEL

We present a light-weight programming model consisting of simple annotations of functions and program variables. Each annotation is designed to be unambiguous and to be automatically testable (e.g. during debugging runs). None of the annotations directly triggers parallelization; they only strengthen program analysis.

We propose annotations on function arguments, on functions and on memory variables. Table 1 summarizes the annotations, their semantics and the program analysis they impact.

Annotations like the ones proposed here are already used frequently in compilers with the goal of conveying additional semantic information that enables or disables specific optimizations, forces particular code generation schemes, etc. Such annotations however are selected for a particular purpose. As such, annotations already in use have slightly different semantics which make them not exactly right for our purpose.

3.1 Function Arguments

It is well understood that program analysis is incomplete when calls to external functions are made. We propose function argument attributes that help improve alias analysis of external functions.

Attributes for pointer arguments and the return value describe memory accesses. The REF and MOD annotations say whether a pointer argument is used for reading or writing. The NOCAPTURE annotation says that a pointer does not escape through the function, i.e. the pointer is not stored to memory. The KILL annotation indicates that a pointer argument points to a memory region that will be entirely overwritten. This is typically useful for pointers to scalars and structures. When assuming C language semantics, it is generally not possible to guarantee KILL semantics on array arguments because the callee function does not know the size of the array; at best it knows what part of the array it is allowed to overwrite.

These annotations are used by dependence analysis and by data structure analysis (DSA). When examining calls to external functions, these analysis make appropriate assumptions for each function argument, rather than assuming the default that pointer arguments are read, written and escape.

A 5th attribute, RETALIAS(#), specifies that the return value is a pointer that is computed based on a particular pointer argument, identified by #. This attribute is useful for functions that return a pointer to the same memory range as one of their arguments. We have extended data structure analysis to unify data structure nodes for the argument and return value when the annotation is present. This makes DSA more accurate in general.

Table 1: Light-weight programming model annotations

Annotation	Semantics	Influenced analysis
Function arguments		
MOD	Pointed-to memory is modified	Memory analysis
REF	Pointed-to memory is referenced	Memory analysis
KILL	Pointed-to memory is invalidated	Memory analysis
NOCAPTURE	Pointer is not captured (doesn't escape)	Memory analysis
RETALIAS(#)	Return value is pointer and aliases argument number #	Memory analysis
Functions		
SYSCALL	Function may have externally visible side-effects	Dependence analysis
STATELESS	Function does not maintain internal state	Dependence analysis
COMMUTATIVE	Function is commutative	Dependence analysis, code transformation
CONSTRUCTOR(fn)	Function is a constructor, *fn* is the corresponding destructor	Privatization
Variables		
KILL(var)	Statement specifying that *var* is dead	Dependence analysis, privatization

Figure 5 shows the annotated declarations of the *strtok* and *strtok_r* C library functions. The main difference between these functions is that *strtok_r* makes its external state explicit through an additional argument (*save_ptr*). Hereby, the *s* argument is not captured anymore. Also, the KILL annotation indicates that every invocation of *strtok_r* overwrites the memory pointed to by *save_ptr*.

Note that the C language defines the *const* attribute for function arguments. This attribute does not serve our purpose as it is valid C code to cast away the *const* attribute.

3.2 Functions

A function labeled with SYSCALL may execute system calls and thus has externally visible side-effects. Dependence analysis adds mutual dependences between all SYSCALL functions in the PDG, forcing them to execute in original program order.

STATELESS functions do not maintain internal state, i.e. they do not access global variables but they access only data structures included in the argument list. STATELESS differs from GCC's pure and const function annotations as the latter describe functions that do not modify or access memory at all allowing, e.g., common sub-expression elimination of calls to those functions. In our case, we want to indicate that escaped pointers will not be referenced or modified by a STATELESS function call.

The COMMUTATIVE annotation implies that calls to such functions may be reordered, but only one instance of the function may be running at any one time [4]. The commutativity annotation is taken into account by dependence analysis, which modifies call nodes in the PDG by removing memory dependences to data structures that are not included in the argument list. When transforming the parallel code, the function is turned into a critical section by inserting a lock.

The constructor/destructor pair of annotations describe functions that allocate and free data structures. Knowing these functions is particularly important when privatizing data structures, because it is otherwise not possible to duplicate complex (i.e. linked) data structures.

The GCC annotation *alloc_size* is related as it states that the return value is freshly allocated memory, but it does not mention the corresponding destructor. GCC also provides constructor and destructor annotations but these are entirely unrelated as they indicate that the corresponding functions should execute before and after executing main, respectively.

The constructor is a function that returns a pointer to new memory. The constructor may have any set of arguments. When privatizing the data structure, a new call to the constructor will be created with exactly the same arguments as the original constructor call. The destructor is a function with a single argument that is a pointer to the memory to destruct.

For data structures that do not store pointers (a property identified by Data Structure Analysis), we assume that malloc and free are the default constructor and destructor.

3.3 Data Structures

Privatization of data structures is an important prerequisite for enabling parallelization [44]. Privatization is, however, only possible when we know that a data structure is dead, e.g. at the beginning of a loop iteration. Proving this in general is very hard, so we introduce the KILL(var) annotation. KILL(var) is a statement that signifies that the variable *var* is dead at the program point where the annotation is inserted. It is recognized by dependence analysis. The KILL annotation applies to the scalar, array or structure pointed to by *var*. It does not apply recursively to any other data structures referenced by pointers stored in *var*.

4. PROGRAMMING TOOLS

4.1 Discovering Locations for Annotations

The function argument annotations apply foremost to functions external to the current compilation unit, such as library functions but also application functions. While library functions are already annotated by the library writer, it is wise to correctly label the memory semantics of all other externally defined functions. Likewise, constructor/destructor pairs should be labeled. These can be identified by simply tracking calls to allocation and free functions.

Finally, we provide an algorithm for suggesting where to place the KILL attribute on variables, probably the most important attribute of LWPM. The reasoning behind the algorithm is that there exist memory dependence edges in the program dependence graph that are not real; they are included only due to conservatism of compiler analysis. It is possible to identify such candidate edges by comparing the statically computed memory dependences with the dependences measured during a profile run of the program. The dependences observed during profiling are *observed dependences*; the remaining memory dependences are potentially bogus. Several

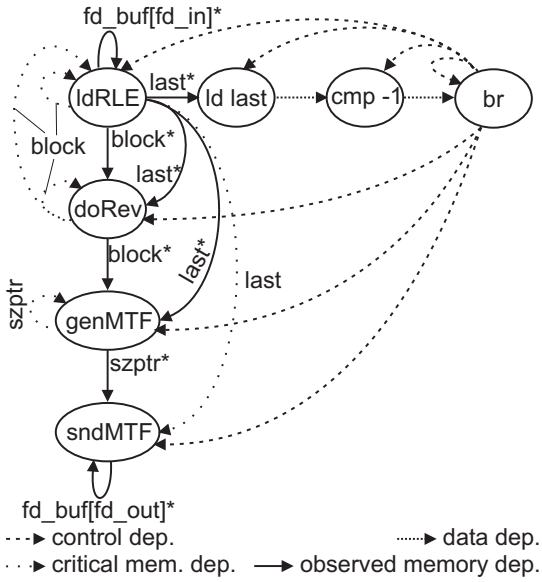

Figure 6: Program dependence graph for the bzip2 example extended with profiled memory dependences.

dynamic dependence profiling tools have been recently discussed in the literature [11, 38, 43]. The algorithm works as follows:

1. In the program dependence graph, memory dependence edges are labeled with the corresponding data structure. This allows us to recognize the data structures that require a KILL annotation.

2. Observed memory dependences are read from the profiling information and corresponding edges are inserted or updated in the PDG. Note that dependences with source and or destination in a callee function must be related to the corresponding call site in the analyzed loop. In the bzip2 example (Figure 6), observed memory dependences are marked with an asterisk.

3. Memory dependence edges in the PDG are re-analyzed. We define a *certain dependence* as either a control dependence, a data dependence or an observed memory dependence. A static memory dependence edge from node M to N is a *critical memory dependence* if there does not exist any path of certain dependences from node M to N. Removing a critical memory dependence may break cycles (split SCCs) and expose more parallelism.

   In the bzip2 example, critical memory dependences exist on the *block*, *szptr* and *last* variables. Removing the critical dependences on *block* or *szptr* will reduce the size of SCCs or turn a self-dependent SCC into a self-parallel SCC. Removing the critical dependence on *last* (from node *ldRLE* to node *sndMTF*) will not break cycles as there is a path of certain dependences between these nodes.

4. For every target node N of a critical memory dependence, we check if there is an observed memory dependence on the same data structure with N as a target. If such a dependence does not exist, then the node N is a potential location for inserting a KILL annotation on the data structure specified in the critical dependence label. If however, such an observed memory dependence does exist, then node N is clearly not a suitable location to insert a KILL annotation.

   We propose inserting the annotation just before a node N in the code, in the same basic block. Alternatively, for a call site node, the annotation may be inserted at the start of the called function.

   These rules lead us to insert a KILL annotation in the example for *block* just before the call to *loadAndRLESource()* and one on *szptr* just before the call to *generateMTFValues()*.

The algorithm produces a list of data structures and program locations where KILL annotations may be appropriate. Data structures are reported by a combination of name (global and local variables) and type. The programmer should verify these annotations and insert those that are correct. We show in the evaluation section that the number of KILL candidates is limited and that they are quite easy to verify.

The algorithm can be made more precise by filtering critical memory dependences. First, if multiple reduction operations are specified on the same variable, then memory dependences appear where the store of one reduction is dependent on the load of a different reduction. These dependences are also not reported, as they are implied by other, likely observed, dependences. Second, we do not propose annotations if either the source or the destination of the critical dependence was not executed during profiling. Third, we do not propose annotations if the data structure is read-only or write-only in the final SCC. Fourth, it is worthwhile to verify that a speedup would be obtained if the annotations were correct: the loop is analyzed as if the annotations were inserted and it is verified that a larger speedup is predicted with the annotations than without.

4.2 Checking Correctness of Annotations

Annotations are meta-information stored in the program source code. As such, they can become incorrect when the source code evolves. To minimize this effect, we designed the annotations to be easily testable automatically, e.g. during debugging or regression runs. Each of the annotations can be automatically turned into a piece of code that is executed during program execution and that tests the validity of the annotation. Most attributes are easy to check (e.g. MOD and REF, NOCAPTURE). Other attributes are specified at the library function level and can be automatically propagated up the call graph (e.g. SYSCALL). The KILL annotation can be checked using methods similar to those used in inspector threads for identifying privatizable data structures [36]. The NOCAPTURE attribute can be tested using dynamic escape analysis mechanisms [29].

5. EXPERIMENTAL EVALUATION

The Paralax compiler is built on top of the LLVM compiler framework. It was specifically constructed to auto-parallelize irregular pointer-intensive programs, such as the SPECint benchmarks.

We demonstrate the efficacy of the Paralax compiler on several benchmarks with coarse-grain parallel loops, taken from the SPECint2000 and SPECint2006 benchmark suites, complemented with clustalw, a bio-informatics benchmark. We selected these benchmarks as they exhibit coarse-grain parallelism. The compiler selects what loops to parallelize based on profiling information and performance models. Other SPECint benchmarks often require speculative parallelization [4], which is not implemented in our compiler.

The benchmark sources are first translated to non-optimized LLVM byte-codes as we generally obtain better results by parallelizing before optimizing. All byte-code files are then linked together in a single file to allow the analysis and code transformation

passes to have a global view of the program (whole program analysis). After parallelizing the code, standard LLVM optimizations passes are run (which are comparable to gcc -O3). We compare speedups relative to the same compilation process without the parallelization step.

Performance is measured on a 2.3 GHz AMD Opteron 2378 quad-core dual processor (shanghai architecture) running Scientific Linux 5.3, kernel version 2.6.18. The LLVM version is revision 83199. Benchmarks are executed on reference inputs.

5.1 Evaluation Results

We compiled the benchmarks using the Paralax compiler assuming that the Paralax compiler knows the annotated declarations for all (used) C library functions, in the style of Figure 5. Furthermore, we used the KILL annotation proposal algorithm to determine such annotations. Dynamic dependence information was captured using the training inputs. Table 2 shows the main loops in the benchmarks and the annotations proposed by our programming tool. Incorrect annotations are shown in italics.

Figure 7 summarizes the performance measurements. In some cases we generate more threads than the number of available cores, as the execution time of some sequential pipeline stages is very low. Still, we map these to their own thread. Loop-speedups are reported in Table 3. These results are discussed next.

5.1.1 Bzip2

Bzip2 is a commonly used (de-)compression utility. The SPECint2000 benchmark repeatedly executes the compress and uncompress steps on an in-memory buffer.

For the bzip2 `compressStream()` function, 9 annotations are correctly identified (Table 2), which must all be added to the program to allow parallelization. The loop is parallelized as a 3-stage pipeline, where the second pipeline stage performs the majority of the work and multiple instantiations may be run in parallel.

The incorrectly identified data structures are related to IO operations, either C library IO data structures or the benchmark-specific structures. The annotation proposal tool identifies these data structures because there were no read-after-write dependences in the dynamic dependence information.

A similar analysis holds for the bzip2 `uncompressStream()` function. Here, an imbalanced 2-stage pipeline with limited parallelism is recognized.

Bzip2 compression speeds up by 2.36 on 8 threads. Speedup stagnates at about 4 threads (Figure 7) which is due to input size restrictions. Also, the parallel-stage pipeline has a quite heavy load in the sequential pipeline stages which take about 35% of the execution time of the pipeline. The speedup of decompression is a mere 1.15 using 2 threads (Table 3). Overall, the speedup is 1.79.

5.1.2 Mcf

The mcf program (SPECint2006) does vehicle scheduling optimization using a simplex graph algorithm. Our compiler recognizes one important parallel loop and identifies a 2-stage pipeline where multiple instantiations of the first pipeline stage can run in parallel. The loop itself is highly parallel allowing a loop speedup of 6.03 on 8 threads. The loop covers only about 60% of the total execution time (Table 3), so overall speedup is limited to 2.06.

5.1.3 Clustalw

The clustalw program performs multiple sequence alignment. The source code is taken from the BioPerf benchmark suite. There are two important phases in the program: pairwise alignment and progressive alignment.

Table 3: Per-loop and overall speedups. The column 'Threads' shows the number of threads used for each pipeline stage.

Benchmark	Loop	Coverage	Best speedup	Threads
bzip2	compressStream	69.4%	2.36	1/6/1
	uncompressStream	29.9%	1.15	1/1
	overall	100%	1.79	
mcf06	primal_bea_mpp	61.2%	6.03	7/1
	overall	100%	2.06	
hmmer	main_loop_serial	99.9%	7.00	1/8/1
	overall	100%	7.00	
clustalw	pairalign	44.5%	4.04	1/8/1
	pdiff	55.4%	1.74	1/1
	overall	100%	2.33	

The `pairalign()` function performs pairwise comparison of a number of DNA sequences, which is trivially parallel. For the Paralax compiler to recognize this parallelism, KILL annotations are necessary on a number of scratch arrays (Table 2). Custom allocation routines are used to create these arrays, so these routines must be labeled as constructors and destructors to allow the compiler to privatize them. Again, KILLs on IO data structures are erroneously proposed.

The `pdiff()` function contains two loop nests that may run in parallel (task parallelism). The loops operate on distinct data structures, so the parallelization transformation does not require privatization of data structures.

Performance of pairwise alignment scales very well with an increasing thread count (Figure 7). The first and last pipeline stages are mapped to their own threads although they are not compute-intensive. Therefore, we can generate code utilizing 10 threads on 8 cores, resulting in a speedup of 4.04. Progressive alignment sees a 1.74 speedup. Overall, the speedup is 2.33 (Table 3).

Higher speedups would be possible for pairwise alignment by utilizing dynamic mapping of iterations to threads instead of static mapping (cf. Section 2.5). Experiments with OpenMP versions of the code confirm this expectation.

5.1.4 Hmmer

The SPECint2006 hmmer benchmark spends virtually all its time in applying a Hidden Markov Model to a set of randomly generated data. Each randomly generated data point can be operated on independently. Extracting this parallelism requires several annotations.

Figure 8 shows a simplified version of the annotated source code of the program's main loop. Two function declarations were annotated. The function `CreatePlan7Matrix()` is annotated to tell the compiler that it is a memory allocation function and that the corresponding deallocation function is the function `FreePlan7-Matrix()`. Also, the function `AddToHistogram()` is labeled as a commutative function: updates to the histogram may occur in any order but each update must run in isolation to avoid data races.

The annotation proposal tool suggests to place KILL annotations on 4 arrays pointed to by mx. These are scratch arrays for the computations made by `P7Viterbi()`. By simple extrapolation we conclude to KILL the entire data structure.

The compiler is able to infer a parallel-stage pipeline with 3 stages. The first stage is self-dependent and is concerned with the random number generation. The second stage is self-parallel and consists of the `DigitizeSequence()`,`P7Viterbi()` and following steps. A final self-dependent pipeline stage is needed for implementation reasons as it hosts loop control and cleanup code.

It is also possible to annotate the random number generator as commutative. This turns the whole loop iteration into a single self-parallel partition (DO-ALL loop). It has, however, the draw-

Table 2: Overview of proposed KILL annotations: the insertion point and the data structure. The number of annotations and the number of correct annotations are shown. Incorrect annotations are shown in italics.

Benchmark loop	Insertion point	Data structures	#Prop	#Corr
bzip2 - compressStream			12	9
	loadAndRLESource	*IO_FILE*, *IO_FILE->buffer*, block, inUse,		
		quadrant, ftab, zptr, *spec_fd_t->buffer*	8	5
	generateMTFValues	unseqToSeq, seqToUnseq	2	2
	sendMTFValues, bsPut*XXX*	*spec_fd_t->buffer*	2	2
bzip2 - uncompressStream			6	5
	getAndMoveToFrontDecode	ll4, ll8, ll16, tt, unzftab, *spec_fd_t->buffer*	6	5
mcf06 - primal_bea_mpp			0	0
hmmer - main_loop_serial			5	5
	P7Viterbi	mx->array (4x)	4	4
clustalw - pairalign			9	7
	forward_pass	HH, DD	2	2
	diff	RR, SS, displ	5	5
	tracepath	*IO_FILE->buffer*	1	0
	fprintf	*IO_FILE->buffer*	1	0
clustalw - pdiff			0	0

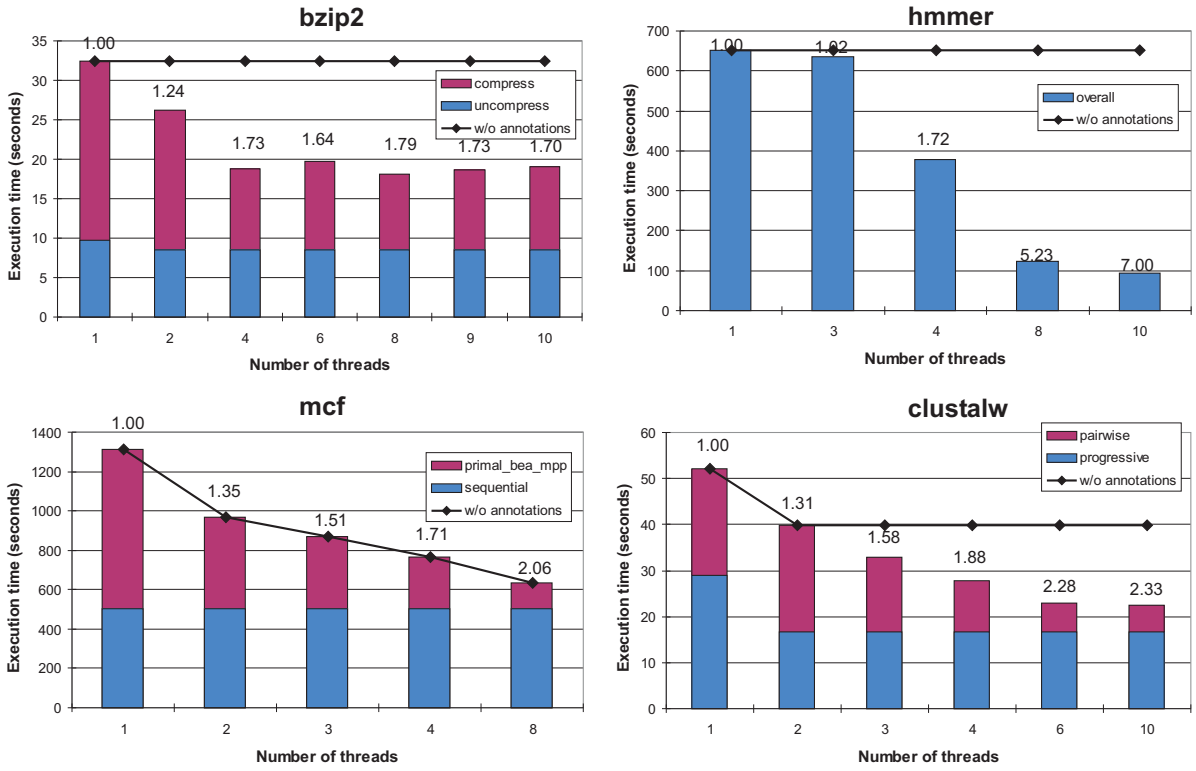

Figure 7: Performance impact of parallelization with and without annotations. The bars show the execution time of the original sequential benchmarks (1 thread) and of parallelizing for multiple threads. Where appropriate, execution time is broken down for different phases. Numbers on top of the stacked bars indicate program speedup. The lines show the execution time obtained when no annotations are added to the program.

back that the randomly generated sequences change from execution to execution, depending on the interleaving of calls to the random number generator. Hereby, the program becomes non-deterministic.

Performance measurements indicate a 7.00x speedup when utilizing 10 threads (Figure 7). Hereto, the version with non-commutative random number generation was used in order to keep execution times comparable.

Note that programmers frequently "optimize" their programs to save on memory allocation time by recycling memory buffers. Had the calls to `CreatePlan7Matrix` and `FreePlan7Matrix` been placed inside the loop body, then the single-threaded execution time would increase by 2%. But, more importantly, the com-

```
/* Modified declarations */
struct Plan7Matrix * CreatePlan7Matrix(...) \
       LWPM_CONSTRUCTOR(FreePlan7Matrix);
void AddToHistogram(...) LWPM_COMMUTATIVE;

/* Parallel loop */
struct Plan7Matrix * mx;
int idx, sqlen;
char * seq, * dsq;
float score;

mx = CreatePlan7Matrix(1, hmm->M, 25, 0);
for(idx=0; idx < nsamples; ++idx) {
   /* Generate random sequence */
   do {
       sqlen = GaussRandom(lenmean, lensd);
   } while(sqlen<1);
   seq = RandomSequence(..., sqlen);
   /* Compute (parallel portion) */
   dsq = DigitizeSequence(seq, sqlen);
   LWPM_KILL(mx->xmx_mem);
   LWPM_KILL(mx->mmx_mem);
   LWPM_KILL(mx->imx_mem);
   LWPM_KILL(mx->dmx_mem);
   LWPM_KILL(mx);
   score = P7Viterbi(dsq, sqlen, hmm, mx, 0);
   /* Update histogram */
   AddToHistogram(hist, score);
   free(seq);
   free(dsq);
}
```

Figure 8: Annotated code of hmmer.

piler would have found the parallelism on its own, i.e. *without* the KILL annotations. The lesson is that recycling memory buffers obscures parallelism; it is an optimization that should be performed automatically after parallelization.

5.2 Discussion

We have have shown that annotations can provide significant speedups, measured on real hardware. These speedups have been obtained with relatively little effort: our implicit parallel programming environment required to validate a small number of annotations, after which an auto-parallelizing compiler extracted the parallelism. If no annotations were inserted, the compiler would not have been able to parallelize the bzip2, clustalw and hmmer benchmarks.

Also, we never instructed parallelization to occur; we merely annotated the programs to explain to the compiler specific details of the programs. It remains the compiler's responsibility to decide when to parallelize and how, based on the estimated speedup of parallelization. Furthermore, it is the compiler who performs all tedious and error-prone tasks such as duplicating variables, creating and synchronizing threads, etc.

This paper presents parallelization results for a relatively small subset of the SPECint benchmarks because auto-parallelization will never succeed on just any program. In fact, the programs must at least contain large independent units of work. This is not the case for the majority of SPECint benchmarks. Some benchmarks might be parallelized by using speculative parallelization techniques [4]. Such approaches will be considered in future work.

6. RELATED WORK

Parallel execution potentially gives important performance benefits. Different approaches to obtaining parallel code have been investigated, giving different levels of exposure of the programmer to the parallelization process.

A myriad of parallel programming languages have been proposed as extensions to sequential programming languages [31, 14] or have been designed for parallelism from first principles [40, 18, 15]. Furthermore, parallel programming models aid in writing parallel code [25, 37]. These approaches however demand *explicitly parallel* programming, which requires significant programming efforts. In contrast, we advocate an *implicitly parallel* approach where the dirty process of parallelization (code transformation, thread mapping, etc.) is performed automatically.

Several systems depend on programmer-supplied annotations for optimization, e.g. the language defined by Guyer and Lin [16]. Annotations describe mod/ref behavior of library functions or they define data transfer functions, describing properties of the computation result conditionally on input values. Many systems use what we call directives, i.e. statements that steer the compiler to perform a specific action. OpenMP pragma's are in this sense directives as they direct the compiler to parallelize a specific loop. Similarly, directives steering parallelization [1] and vectorization [2] are commonly used in explicitly parallel programming languages.

Programmer support environments have been developed to aid the programmer in writing parallel code [3, 20, 21, 23]. These systems are focused on Fortran programs and array-based computations. They are also geared towards explicitly parallel programming languages. In contrast, this paper considers a different application domain and implicit parallel programming.

Compile-time automatic parallelization has been successful on array-based code, leveraging DOALL and DOACROSS parallelism [5, 22, 30]. These techniques fail however on irregular pointer-based applications, due to the absence of significant loops performing array-based computations. In this paper, we show that auto-parallelizers must be structured differently for this type of code and must search for different types of pipeline parallelism at coarser levels of granularity.

It has been proposed to apply speculative parallelization on irregular pointer-based applications [7, 17], but these systems typically require hardware support. CorD [42] is a software-only speculative parallelization technique. However, they always parallelize speculatively, even if the parallelism is non-speculative. Decoupled software pipelining [33] is another approach for parallelizing irregular pointer-based applications at a fine-grain level. Both compiler and hardware support are assumed. The Galois system is a programming model supporting irregular data-parallel applications [26]. They too rely on speculative execution but it is the programmer who identifies when to speculate.

Bridges *et al.* [4] study the performance of manually identified speculatively parallel code regions. Using trace analysis and extrapolation of performance, they obtain quite reasonable speedups, which vary greatly between benchmarks. Their approach is not practical; they present an estimate of potential speedup given extensive hardware and software support.

Dynamic parallelization is performed at runtime based on input data [39]. Data dependences are dynamically profiled and/or checked before executing in parallel [8, 34]. Software behavior oriented parallelization [10] allows the programmer to identify possibly parallel code regions and uses a runtime system for speculatively parallel execution. In [41], dynamic dependences are used to decide on the correctness of pipeline parallelism at runtime. In contrast, we rely on static compile-time parallelization. Dynamic

dependence profiling can aid the programmer in inserting annotations, but it is not essential. Thus, we avoid runtime overheads.

In summary, our work is unique as it targets irregular pointer-intensive codes, presents an auto-parallelizing compiler for such applications and assumes an implicit parallel programming model. Furthermore, it presents programming tools for proposing and testing program annotations in the context of an implicit parallel programming model.

7. CONCLUSION

This paper describes an implicit parallel programming environment geared towards irregular and pointer-intensive applications such as the SPECint benchmarks. In implicit parallel program, the goal is to write a sequential program (yielding the relative simplicity of developing non-parallel programs) that can be automatically parallelized with important performance improvements.

We present the Paralax compiler, an auto-parallelizing compiler that is constructed specifically for parallelizing irregular pointer-intensive applications. Hereto, we focus on coarse-grain dependence analysis and coarse outer program loops. We show that substantial parallelism exists at this level.

In order to aid the Paralax compiler in finding significant thread-level parallelism, we present a light-weight programming model to fill in the semantic gaps. The light-weight programming model adds annotations to a program that describe well-defined properties of functions, variables and data structures; information that a static compiler cannot infer. The annotations are designed such that verification of their correctness is fairly easy.

Furthermore, we present programming tools to support implicit parallel programming: automatically testing the correctness of annotations during debugging runs and automatically proposing annotations based on dynamic dependence information. This helps to upgrade a sequential program to an implicitly parallel one.

Application of our implicit parallel programming environment to the SPECint benchmarks shows promising results. On a dual processor system with two quad-core processors, we demonstrate overall program speedups in the range of 1.79 to 7.00 when using 8 cores, even though some benchmarks have limited parallelism.

In future work, we plan to improve the auto-parallelizing compiler by utilizing speculative parallelism and by adding intra-data structure dependence analysis.

8. ACKNOWLEDGEMENTS

The authors are grateful to Albert Cohen and the anonymous reviewers for their insightful comments. Hans Vandierendonck is a Postdoctoral Fellow with the Fund for Scientific Research–Flanders. Sean Rul is supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT). This research was also sponsored by Ghent University and by the European Network of Excellence on High-Performance Embedded Architectures and Compilation.

9. REFERENCES

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications, Inc.,/McGraw-Hill, Inc., New York, NY, USA, 1992.

[2] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[3] T. Brandes, S. Chaumette, M. C. Counilh, J. Roman, A. Darte, F. Desprez, and J. C. Mignot. Hpfit: a set of integrated tools for the parallelization of applications using high performance fortran. part I: HPFIT and the TransTOOL environment. *Parallel Comput.*, 23(1-2):71–87, 1997.

[4] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, 2007.

[5] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 21(7):162–175, 1986.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, 2006.

[8] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446, 2003.

[9] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267, 1996.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234, 2007.

[11] K.-F. Faxén, K. Popov, S. Jansson, and L. Albertsson. Embla: Data dependence profiling for parallel programming. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 780–875, 2008.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[13] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, 2000.

[14] M. Frigo, C. E. Leierson, and K. H. Randall. The implementation of the Cilk-5 multi-threaded language. In *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation*, pages 212–223, 1998.

[15] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, 2002.

[16] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 4–4, 1999.

[17] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on*

Architectural support for programming languages and operating systems, pages 58–69, 1998.

[18] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, 2003.

[19] W.-M. e. Hwu. Implicitly parallel programming models for thousand-core microprocessors. In *Design Automation Conference (DAC-44)*, 2007.

[20] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 244–251, 1991.

[21] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP. *IEICE - Trans. Inf. Syst.*, E89-D(2):399–407, 2006.

[22] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[23] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):329–341, 1991.

[24] K. Knobe and V. Sarkar. Array SSA and its use in parallelization. In *25th Annual Symposium on the Principles of Programming Languages*, pages 107–120, Jan. 1998.

[25] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[26] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[27] C. Lapkowski and L. Hendren. Extended ssa numbering: Introducing ssa properties to langauges with multi-level pointers. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 23. IBM Press, 1996.

[28] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, 2007.

[29] K. Lee and S. P. Midkiff. A two-phase escape analysis for parallel java programs. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 53–62, 2006.

[30] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, New York, NY, USA, 1997. ACM.

[31] A message passing interface standard, version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, Sept. 2009.

[32] D. Novillo. Memory SSA - a unified approach for sparsely representing memory operations. http://www.airs.com/dnovillo/Papers/mem-ssa.pdf.

[33] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[34] P. Peterson and D. A. Padua. Dynamic dependence analysis: A novel method for data dependence evaluation. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 64–81, 1993.

[35] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, 2008.

[36] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 137–146, 1995.

[37] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (STAPL). In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 402–409, 1998.

[38] S. Rul, H. Vandierendonck, and K. De Bosschere. Extracting coarse-grain parallelism from sequential programs. In *International Conference on Principles and Practices of Parallel Programming*, pages 281–282, Feb. 2008.

[39] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 263–273, New York, NY, USA, 2007. ACM.

[40] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–271, 2007.

[41] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[42] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, 2008.

[43] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 177–187, jun 2009.

[44] P. Tu and D. A. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, London, UK, 1994. Springer-Verlag.

[45] H. Vandierendonck, S. Rul, and K. De Bosschere. Factoring out ordered sections to expose thread-level parallelism. In *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, page 8, June 2009.